Project 2: User Programs

Preliminaries

> Fill in your name and email address.

罗兆丰 2100012985@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Argument Passing

DATA STRUCTURES

> A1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

static void push(char **sp, char *arg, int len);

a function to push a given argument onto the stack, by moving the stack pointer and writing the argument's contents.

No new global/static variables or structs were added, but the following local variables were added in the start_process function:

int argc = 0;

to keep track of the number of arguments passed to the process.

void *argv[114];

an array to hold the addresses of the arguments passed to the process.

char *token;

used to store the tokens generated by strtok_r when parsing the command line arguments.

ALGORITHMS

> A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?

Argument parsing was implemented using the strtok_r function, which was used to split the command line arguments into separate tokens. Each token was then pushed onto the process's stack using the push function, and the address of each argument was added to the argv array in reverse order.

To avoid overflowing the stack page, the push function checks whether the stack pointer is still within the limits of the stack page before writing to it. If the stack pointer exceeds the stack page limit, the process is

terminated with an exit code of -1.

The number of arguments is limited to 114, so if the number of arguments exceeds this limit, the process is terminated with an exit code of -1.

RATIONALE

> A3: Why does Pintos implement strtok_r() but not strtok()?

Pintos implements strtok_r instead of strtok because strtok is not thread-safe. strtok uses static state to keep track of the position in the input string, which means that if multiple threads call strtok simultaneously, they can interfere with each other's state and produce incorrect results. In contrast, strtok_r is thread-safe because it uses a pointer to a private state variable that is passed between calls.

> A4: In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

There are at least two advantages of the Unix approach to separating commands into an executable name and arguments, which is typically done by the shell:

Flexibility: The Unix approach allows for greater flexibility in how commands are constructed and executed. Users can easily compose complex commands using pipes, redirects, and other shell constructs, which would be more difficult to achieve if the kernel had to parse and interpret the entire command line. This flexibility also allows for greater customization and scripting of system behavior, which is a hallmark of Unix-like systems.

Separation of concerns: Separating the command-line parsing and interpretation from the kernel allows for a clearer separation of concerns between the kernel and user-space processes. The kernel can focus on managing system resources and enforcing security policies, while the shell can handle user input and construct the appropriate commands to execute. This separation also allows for easier replacement of the shell or other user-space utilities, without affecting the underlying kernel implementation.

System Calls

DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

struct child_info

used to record information and exit_code of child processes in the parent process, as well as blocking the parent process when calling wait.

{ struct thread *t;

bool is_alive;

tid_t tid;

int exit_code;

struct semaphore sema;

bool been_waited;

struct list_elem elem;

};

struct file_info

used to record each open file and its corresponding file descriptor.

{

struct file *f;

int fd;

struct list_elem elem;

};

In struct thread:

int exit_code; the exit value of the process

struct thread *parent; the parent process of the thread.

struct list child; a list of child processes' information.

struct child_info *info; the child process information when the thread is a child process.

bool child_exec_success; whether the child process is successfully executed.

struct semaphore sema_exec_child; a semaphore used to ensure that the child process returns only after it has been confirmed to have succeeded or failed.

int last_fd; the last assigned file descriptor.

struct list file_list; a list of open files.

struct file *executing_file; the file that is currently being executed, to reject writes to it.

> B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

File descriptors are associated with open files through a per-process file descriptor table. Each time a file is opened, a file descriptor is assigned to it, which is then used to access the file through the file descriptor table. The file descriptor table is specific to each process, so file descriptors are unique only within a single process.

ALGORITHMS

> B3: Describe your code for reading and writing user data from the kernel.

When reading or writing user data from the kernel, the main concern is to ensure the validity of the user pointer. To address this, three functions are used to check the validity of the pointer:

static void check_valid_addr(const void *ptr, int size);

static void check_buffer(void *buff, unsigned size);

static void check_string(void *str);

The check_valid_addr function checks whether a pointer ptr pointing to size bytes of memory is valid. It checks size times whether the pointer is NULL, points below PHYS_BASE, points above 0x8048000, and whether it points to a valid page within the page directory of the current process.

The check_buffer function checks whether an entire buffer of size bytes is valid by repeatedly calling check_valid_addr on each byte in the buffer.

The check_string function checks whether a string is valid by repeatedly calling check_valid_addr on each byte until a null terminator is encountered.

> B4: Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to pagedir_get_page()) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?

For a system call that copies a full page (4,096 bytes), the minimum number of inspections of the page table required is 1, and the maximum number is 4,096. The minimum is achieved when the data is aligned, and the maximum is achieved when every byte of the page must be checked to ensure that it is valid.

For a system call that copies only 2 bytes of data, the minimum number of inspections of the page table required is also 1, and the maximum number is 2.

To optimize these numbers, it is possible to calculate the first and last bytes of the data to be read/written and then perform only one page table access for each of the pages that contain these bytes, as well as any pages in between. This way, at most two page table accesses are required, regardless of whether the data size is 4,096 bytes or 2 bytes. This is an effective optimization method.

> B5: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

The "wait" system call is implemented by calling the process_wait function. Within process_wait, the child processes are iterated over to find the one with the matching tid. If it is not found, -1 is returned. If it is found, been_waited is checked to see if the child process has already been waited on. If it has, -1 is returned. If the child process has already exited, its exit code is returned. Otherwise, sema_down is called to block the parent process and wait for the child process to return.

When a child process exits, its information structure is updated with its exit code. If the parent process is waiting for the child process, sema_up is called to wake up the parent process.

In this way, the parent process is synchronized with the child process and can receive its exit code when it is ready.

> B6: Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.

Strategy:

1、Allocate and initialize all necessary resources (such as locks, buffers, etc.) in advance, and release these resources promptly when errors occur to avoid resource leaks.

2、Add error checking and handling logic in the code to minimize the chance of program crashes caused by accessing user-specified memory.

3、Use the three pointer checking functions mentioned above to ensure the legality of the memory access.

For example, when implementing a function that reads user-specified memory, we can use the following strategy:

Allocate and initialize necessary resources (such as locks and buffers) at the beginning of the function.

Before accessing the user memory, use the pointer checking functions mentioned above to catch potential errors.

If an error is detected, immediately terminate the current process.

In the thread_exit function, release all allocated resources and return the error to the caller.

If the user memory is read successfully, terminate the thread at the end of the function and release all allocated resources.

By following these strategies, errors in the code can be managed more effectively, while maintaining code readability and robustness.

SYNCHRONIZATION

> B7: The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?

To ensure this, we use a semaphore to block the parent thread until the child thread has finished loading the new executable. Specifically, when the process executes the "exec" system call, it creates a new child thread and then blocks itself by executing sema_down. The child thread then executes the function start_process(), which loads the new executable. If the loading is successful, the child thread sets the parent's child_exec_success flag to 1 and then executes sema_up to unblock the parent thread. The parent thread can then check the value of child_exec_success to determine if the new executable was loaded successfully.

If the loading is unsuccessful, the child thread sets child_exec_success to 0 and executes sema_up to unblock the parent thread. The parent thread will then return -1 to indicate that the "exec" system call failed.

> B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?

To keep track of the state of child processes, the parent process can use a child_info structure to store information such as whether the child has exited and its exit code. When a child process exits, it modifies its corresponding child_info structure to notify the parent process of its exit status. The memory used for child_info can be freed in thread_exit. There are four possible scenarios to consider:

The parent process calls wait(C) before C exits: In this case, the parent process blocks itself, and C wakes up P when it exits. The parent process can then obtain the exit code from the child_info structure.

The parent process calls wait(C) after C exits: In this case, the parent process can directly obtain the exit code from the child_info structure.

The parent process terminates without waiting, before C exits: In this case, the child process detects that the parent process has terminated and can free the child_info structure.

The parent process terminates without waiting, after C exits: In this case, the parent process detects that the child process has exited and can free the child_info structure.

Additionally, there is a situation where the parent process waits for the same child process multiple times. In this case, the exit code can be set to -1, and the parent process can exit.

RATIONALE

> B9: Why did you choose to implement access to user memory from the kernel in the way that you did?

The way I chose to implement access to user memory from the kernel is to check whether the pointer is null or falls below PHYS_BASE or above 0x8048000, and then use pagedir_get_page to confirm whether the pointer falls within the page table allocated to the process.

The advantage of this approach is that it is simple and easy to implement.

> B10: What advantages or disadvantages can you see to your design for file descriptors?

The advantage of my implementation is its simplicity, where file descriptors are allocated in a monotonically increasing manner without the possibility of allocation errors or duplicates.

The disadvantage is that every time a file descriptor needs to be looked up to access the corresponding file, the file descriptor list must be traversed, resulting in a time complexity of O(n), which could be significant for large numbers of open files.

> B11: The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

I did not change the default tid_t to pid_t mapping because in Pintos, one process corresponds to only one thread. However, changing the mapping could allow for one process to correspond to multiple threads,

enabling more advanced thread management and communication patterns.